An Architectural Blueprint for a Pluggable Version Control System for Software (Evolution) Analysis

Ghezzi, Giacomo ; Würsch, Michael ; Giger, Emanuel ; Gall, Harald C

DOI: https://doi.org/10.1109/TOPI.2012.6229803

An Architectural Blueprint for a Pluggable Version Control System for Software (Evolution) Analysis

Giacomo Ghezzi, Michael Würsch, Emanuel Giger, and Harald C. Gall
Department of Informatics
University of Zurich
Zurich, Switzerland
{*ghezzi,wuersch,giger,gall*}*@ifi.uzh.ch*

***Abstract*—Current version control systems are not built to be systematically analyzed. They have greatly evolved since their first appearance, but their focus has always been towards supporting developers in forward engineering activities. Supporting the analysis of the development history has so far been neglected. A plethora of third party applications have been built to fill this gap. To extract the data needed, they use interfaces that were not built for that. Drawing from our experience in mining and analyzing version control repositories, we propose an architectural blueprint for a plug-in based version control system in which analyses can be directly plugged into it in a flexible and lightweight way, to support both developers and analysts. We show the potential of this approach in three usage scenarios and we also give some examples for these analysis plug-ins.**

***Keywords*-software evolution; version control systems; mining software repositories**

I. Introduction

The concept of managing the subsequent versions of the source code of a software project, and any other related document, has been around since the dawn of software development. The actual concept of a version control system (*VCS*) and its first implementation was introduced by Rochkind [1] in the seventies. Systems belonging to this first generation were file-oriented, centralized, locking-based and without network access capability.

CVS [2], an evolution of RCS [3] paved the way for a second generation. It was explicitly designed for collaborative development and used a merging rather than locking-based approach. Through a client-server mode, geographically scattered developers were supported in working as a team.

The third and most recent generation represents yet another major conceptual shift: native complete decentralization. These systems quickly gained a remarkable popularity and ground over older, centralized systems, as dispersed, Internet-mediated software development became the norm rather than the exception. Well-known representatives of such distributed *VCSes* are Git and Mercurial.

As this very concise history shows, *VCSes* greatly evolved since their introduction. However, regardless of different implementation details and features provided, their core functionality and rationale never changed. No matter what *VCS* is being used, there are three basic things a user can do; *check out* a file copy from a repository, *check in* or *commit* a change on a file to its master in a repository, and *view the history* of files. Everything else is an elaboration or support for these three operations.

While the information stored in versioning systems supports traditional forward engineering activities sufficiently well, it is not complete enough to perform comprehensive evolution analysis or reverse engineering. Recent research, however, has shown that there is much to be learnt from the development history of programs. The lack of support for such analyses has been filled so far by third-party tools that exploit *VCS* repository data to extract all sort of information, e.g. logical couplings, source code metrics, evolution of code clones, potential bugs, etc.

The only way such tools can retrieve this data has been the parsing and analysis of the bare history log; which is, in our opinion, far from being the optimal approach. These logs, in fact, record the history of a repository in a synthetic way and are meant mostly for users to keep track of the development history. Incremental, proactive processing is barely supported and retro-active computations are long, resource-intensive and often error prone. To put it in a nutshell, current *VCSes* are not built to be systematically analyzed. This had a negative impact in the adoption of many software evolution analyses. In order for them to play a part in the developers' day-to-day processes and to prove their immediate usefulness, a more incremental, lightweight and integrated approach is needed, as pointed out by Zeller [4].

In this paper we propose a plug-in-based *VCS*. Different analyses can be plugged into the system and register for specific repository events (e.g., a commit, the tagging of a new release, etc.). In this way, they can automatically and proactively run and update their data every time it is needed in an incremental fashion. Moreover, with the data produced, analyses can enrich the limited *VCS* data already existing. In the remainder of this paper, we first describe our proposed architectural blueprint. Next, we discuss the benefits of the architecture by comparing it to existing approaches in the context of three software evolution analysis scenarios. We then conclude with a brief discussion on future work.

II. Architecture Overview

Figure 1 gives a quick overview of the proposed architecture. At its core remains a standard *VCS*, offering all the functionalities of any modern, state of the art system. In fact, it is not our goal to propose a brand new *VCS*, as the current generation already supports forward engineering activities sufficiently well. Instead, we aim to enhance the existing ones by building a lightweight plug-in architecture on the top of them to remedy the lack of evolution analysis support in a flexible and transparent way. Most of the current

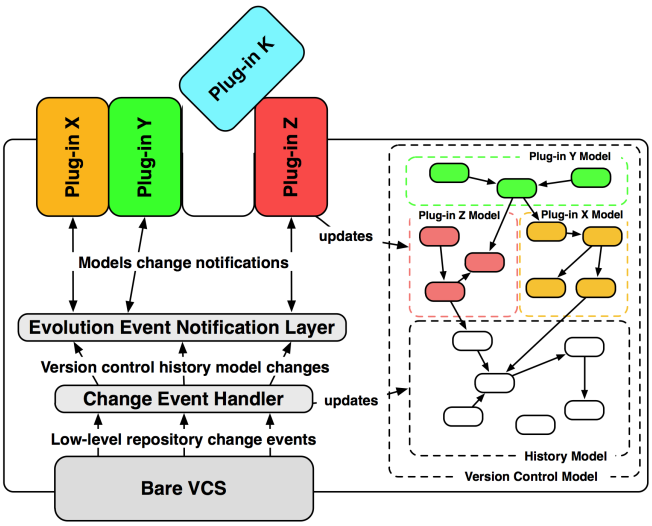

Figure 1. Overall view of the envisioned architecture.

VCSes already offer a mechanism to perform automated actions in response to specific events occurring in a repository. These actions are commonly called *hooks* or *triggers*. As of now, these mechanisms are underused and only for rather simple, low level tasks such as checking the compliance of commit messages, sending mail notifications, etc. In our architecture, these events are caught by the *Change Event Handler*, which uses them to build and maintain a detailed, high-level model of the history and state of the repository. This model, which we call *Version Control History Model*, describes all the essential concepts of a project's version control, independently of the actual *VCS* used. This is possible because most of the major version control systems share the same conceptual model, with just some slight differences in terminology. This is the model that the plug-ins see and use to fetch all the repository data they need. Once these events are processed and the model has been updated, an event containing the detailed change information is published to the *Evolution Event Notification Layer* which will notify the plug-ins. They will then use this information to run their analysis and/or update their data. Apart from consuming specific events, plug-ins can also directly query the model to extract further data. In fact, since an event only contains information on the entities directly involved in it, information about the past needs to be fetched directly from the model.

In the following we briefly describe each of the architectural components.

A. Change Event Handler

The *Change Event Handler* is the low level component through which our architecture connects to the actual *VCS*. *VCSes* produce several types of events, however, in our case we only catch the following:

- **Commit** occurs when a new change set is successfully committed in the repository.
- **Tag** occurs when a tag (also known as release) is created.
- **Branch** occurs after a new branch is created.
- **Merge** occurs when two branches or a branch and the main development trunk are merged together. It is basically a special commit.

The *Change Event Handler* extracts from these low level events all the information necessary to build and maintain the *Version Control History Model*. A high-level event reflecting the changes in the model is then created and published to the *Evolution Event Notification Layer*. These events contain the information about all the involved model entities. They can be considered a sort of translation of the repository events into a format that can be understood and used by the registered plug-ins.

B. Version Control History Model

The *Version Control History Model* is split into different parts. At the core of it lies the *History Model*, which describes all the core concepts of a version control history and is the part being managed by the *Change Event Handler*. This is the only part that is built into the system by default. Plugins can then expand and enrich it by defining on top of it their own sub-models to describe their analysis data. The main concepts of the *History Model* are:

- **Release** represents a snapshot in time of the project, labeled with a meaningful name or number. It thus comprises all the involved files versions consistently with the time of the snapshot (the most recent version, given the time of the snapshot). A new one is created whenever a *tag* event is caught.
- **Branch** represents a branch of the project codebase at a specific point in time. In this way, both the branch and the original development stream can be worked on and evolve independently. A new one is created when a *branch* event is caught.
- **File** represents any file being tracked in the repository.
- **Version** is also known as Revision, it represents any type of change to a file under version control that was committed to the repository. It is uniquely associated with the file involved in the change. A new one is

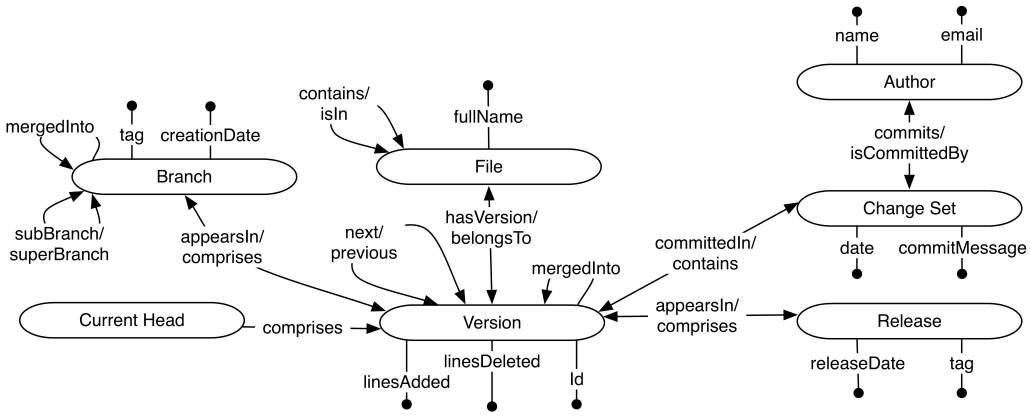

Figure 2. Overall view of the version control model.

created whenever a *commit* event is caught, for every file involved.

- **Change Set** represents the set of changes on files that are written back to the repository at a specific moment in time by a user. It results in the creation of a new version of each modified file. A new one is created whenever a *commit* event is caught.
- **Author** represents a committer of the project.
- **Current Head** represents the current state of the main development trunk or stream. That is, the most recent versions of all its files.

Figure 2 gives an overview of this model. Hiding the repository events behind the *Change Event Handler* and using such a high-level model makes the entire architecture extremely flexible. In fact, it can be deployed on top of most *VCSes* by just providing a different *Change Event Handler* built to handle and interpret the system specific events. Moreover, the model structures repository data in a more intuitive and logical way. It can be directly queried by plug-ins, without them having to either analyze the repository history logs or its internal, low-level representation. Making the model extensible allows plug-ins to seamlessly enrich the original version control history data with their analysis data. Furthermore, it also enables them to benefit and use other plug-ins' data, thus supporting analyses not just based on the core historical data, but also on additional analysis data.

We define our model as an ontology with the Web Ontology Language OWL [5] for three main reasons. First, a model described with such technology exhibits explicit semantics and is much more flexible to changes than one backed by a relational database. For example, it is unproblematic to extend ontologies by additions or by specializing existing concepts. On the other hand, a change in a traditional database-backed model would usually require schema changes, which is a time consuming operation. Existing applications already accessing the database would likely break subsequent to the change. Second, ontologies were explicitly designed to be shared. They can be serialized using the RDF/XML standard and exchanged—in our case with the registered plug-ins—without any loss of data semantics. Third, a powerful and standardized language, SPARQL [6], can be used for querying.

C. Evolution Event Notification Layer

This component is in charge of sending the high-level events to the plug-ins. It is based on a publish-subscribe pattern, in which the subscribers are plug-ins and the publishers are the *Change Event Handler* and the plug-ins. Plugins can register for one or more types of events and react accordingly.

By default, only the four basic event types (*commit*, *tag*, *branch* and *merge*) are maintained by the system. Plugins, however, can register additional ones to publish any information about changes to their analysis data and thus notify any other possible plug-in that consume their data.

D. Plug-Ins

Plug-ins are the consumers of the aforementioned events. They also query and extend the model while running analyses or offering additional functionality that extends and enriches that of the base *VCS*. Events only contain information on the entities directly involved in it. Further information, e.g., about the past of the entities then needs to be fetched from the model.

For example, a plug-in calculating version control history metrics, such as number of committed lines of code for each developer, activity by clock time, or the growth of the project's total lines of code over time, would just need the information about the new commit to update its data. On the other hand, a plug-in running a source code analysis, might need to get a snapshot of the entire project taken on the date of the new commit. Plug-ins do not necessarily just passively consume version control data for their own purpose. They can also expand the *Version Control History Model* with their own sub-model describing the data they produce. In

this case, they can then also register new events representing the changes in their model and publish them to the *Evolution Event Notification Layer*.

E. An Operational Example

What follows is a quick outline of how our architecture and, in particular, its components react to a standard commit to the repository. A standard commit event is issued by a *VCS* every time changes on tracked files are successfully written back to the repository in an atomic operation. It always consists of a list of modified, added or deleted files, its unique version control ID or number and a message written by the author of the changes to describe them. Some systems also provide the number of added and deleted lines for every file involved (e.g. GIT), whereas others do not (e.g SVN). When the *Change Event Handler* catches a change event, a new *File* is created for every file that had just been added to the repository. A new *Version* is then created for each changed file and linked to its related *File*, as well as to the most recent previously committed *Version* (if there is any). If the committer does not yet exist in the model, a new *Author* is created, using all the information that can be extracted from the event. The *Versions* and the *Author* are then linked to a new *Change Set* representing the commit. The set will also contain the commit message, date and ID. Figure 3 shows the entities created and updated after a very simple commit. A new *commit* event with all these involved entities is then published to the *Evolution Event Notification Layer*.

III. Usage Scenarios

In the following, we list three problem scenarios in the context of software evolution analysis. For each scenario, we present existing solutions and their shortcomings; we then outline how our approach is able to overcome them with different plug-ins.

Scenario 1: Continuous Code Quality Check

Description: *Several studies proved that source code metrics are beneficial to steer the software development lifecycle. They are usually used to assess its overall quality [7], [8], discover problematic entities [9] or predict defects [10]. Currently, VCSes only save and keep track of files. They do not discern between the different file types nor they analyze them. This means that, to calculate metrics, the entire source code needs to be fetched from the repository and parsed or even partially compiled.*

Existing Approaches: As of now, these metrics are calculated using third party tools. A snapshot of the project is manually checked out on a local machine and its source code fed to the metrics calculator of choice. Many IDEs have integrated calculators so that developers can check the metrics on their copies whenever needed. This approach however works only on local copies of the source code. Web-based software quality platforms, e.g., Sonar,[1] partially automate this process by fetching the source code to analyze directly from the repository upon user request. These systems can be triggered by Ant, Maven or Continuous Integration servers. This type of solution has proven to be highly successful and it is a big step in the direction of easy, automated software analysis. However, it requires installation and set up of a separate stack of heavy-weight applications, even for very simple measurements. Such an approach is also still not proactive, as the calculations have to be manually triggered by a user or a tool (e.g., a continuos integration application). Furthermore, if additional code quality indicators such as Code Smells or Disharmonies [7] need to be calculated, the exact same metrics will probably be recalculated again and again. In fact, all these tools are written to be used on their own and not to be combined or to share their data with each other. Even though these quality indicators are calculated using the exact same metrics already extracted, these synergies are lost.

Our Approach: A metrics plug-in is the only thing needed to address this scenario. This plug-in subscribes to *commit* events published by the *Evolution Event Notification Layer* and proactively, continuously updates or calculates its metrics. Thus, at any point in time, the metrics data is always up to date. The speed of the analysis would also benefit. In fact the calculator works on very little changes (every commit) and accesses the files to analyze locally, without having to fetch them remotely and incurring in additional overhead. This plug-in could then also attach its own sub-model to the *Version Control History Model*, to describe the metrics it calculates and to relate them to the files under version control. In this way, an additional Code Disharmonies calculator could exploit this information to quickly keep track of all the suspect files exhibiting smells such a God Class, Brain Class, etc. Software engineers could either monitor these metrics by means of a Web front-end or with a plug-in for their IDE.

Scenario 2: Extracting Fine Grained Source Code Changes

Description: *VCSes still keep track of changes in a simplistic way, storing just the text lines that were added and/or deleted. Fine grained structural changes in the source code are not considered at all. Developers have to rely on textual diffs to really understand what and how code entities changed between different versions. Several studies already showed the usefulness of extracting and using such changes to detect re-factorings, discern different significance level of changes, better predict bugs, etc.*

Existing Approaches: In most of the cases, source code changes are extracted "a posteriori" given the entire *VCS* history for historical analysis [11], [12]. This is extremely

[1] http://www.sonarsource.org/

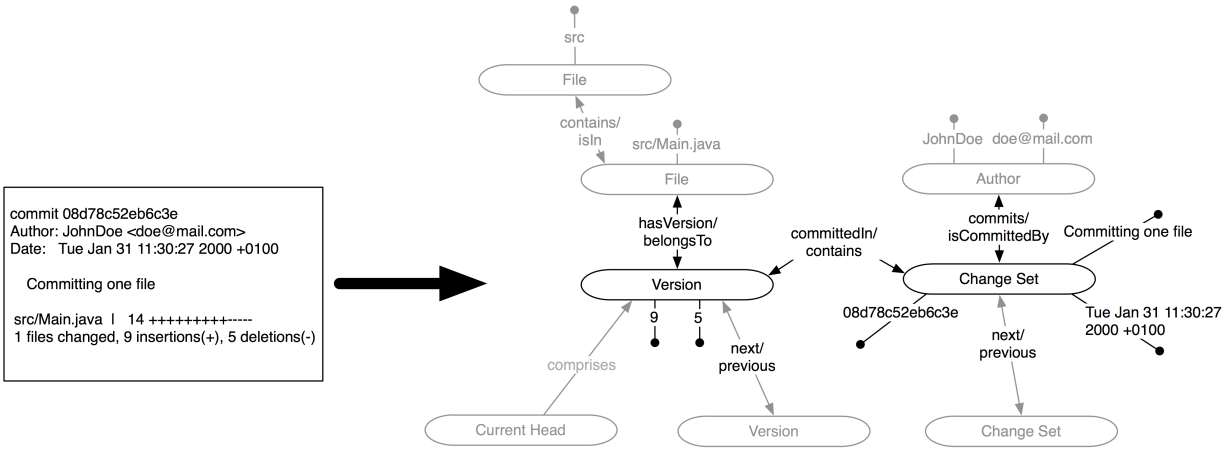

Figure 3. Handling of a new commit event.

time consuming, as every single revision of every file has to be fetched from the repository and parsed to extract the information required. Moreover, these tools are not automatically triggered by changes in the *VCS* repository but have to be manually executed. Other tools have taken a more automated/proactive approach, extracting that information as changes are performed on a developer's local machine [13] or as they are committed to the repository [14]. So far, all these existing solutions are based on research prototypes and have not been incorporated in any of the commonly used *VCSes*.

Our Approach: A change extractor plug-in incrementally extracts these changes every time a new commit is performed. This plug-in responds to commit events through the *Evolution Event Notification Layer*. Every time one is received, it fetches the content of all the files involved and of their previous versions and calculates the fine-grained changes. In this case, our system works much like Molhado [14], while being based on any already existing, well-known and widely used *VCSes*. Similar to the previous scenario, this incremental, proactive extraction of data is highly beneficial in terms of performance and ensures always up-to-date data.

Scenario 3: Flexible Querying of Custom Data

Description: *Modern VCSes do not have an interface through which information about them and their history can be programmatically extracted. The only way is to analyze their history logs, which have mainly been devised for record-keeping. They are intended to be read by human users and not suitable for systematic analyzes. Moreover, their format and syntax depends on the actual VCS. This means that every analysis, not only has to parse and interpret the log, but has to do that for every VCS addressed.*

Existing Approaches: All the major *VCSes* offer web interfaces both natively or through third-party tools. With these interfaces it is possible, for example, to see a list of all the files changed, added or deleted in any given revision or to compare two versions of a file manually to see what has been changed. These interfaces help human users to navigate the repository. An equivalent interface for applications, through which the repository can be queried for information, is still missing. Tappolet et al. [15] introduced the concept of semantics-aware, queryable *VCSes*. However, to the best of out knowledge, it has never been implemented.

Our Approach: A *SPARQL Endpoint* plug-in allows users to query the internal version control history model with SPARQL [6] queries. This plug-in obviously needs to define and publish an ontology to describe that model, or use an existing one such as the ones introduced in [16]. This is necessary, so that users know the exact semantics and thus are able to write valid, meaningful queries. The plug-in then translates the SPARQL queries it receives into internal queries to fetch data from the internal model and then translate the results back into SPARQL Results [17]. In this way, from a user's perspective, the repository acts exactly like a RDF/OWL triple store. With our approach the query possibilities are manifold. Different query interfaces for different languages could be plugged into the system. For example, another plug-in could allow user to query the repository with natural language following the approach proposed by Würsch et al. [18].

IV. Related Work

The idea of extending the functionality of a *VCS* is not new. Most of the currently existing systems already offer that. These extension mechanisms range from simple scripts only activated by specific repository events (e.g. SVN) to more complex plugin style solutions (e.g. Mercurial and Bazaar). These extensions can do a variety of things, including overriding commands, adding new commands, providing additional network transports, customizing log output, adding an alternative diff algorithm, etc. However, they are always aimed at extending or customizing the core functionalities of those systems. Our solution is not aimed at enriching those

functionalities, but rather at building an infrastructure on top of a standard *VCS* to support its analysis. To the best of our knowledge such a solution has not been proposed yet.

There is a plethora of tools and frameworks exploiting software project data for all sorts of software evolution analysis. However, none of them are integrated within *VCSes*. Most of them, such as for example CodePro Analytix[2], require the installation of tools on a local machine and the manual triggering of such analyses. Sonar represents a step into a much more automated and continuous, plugin-based analysis engine for software projects. Nonetheless it is still not integrated with the targeted *VCSes* and it is mostly focused on software analysis (code coverage, test coverage, clone detection, etc.) and not on evolution.

We share with Molhado [14] the concept of an extensible, logical representational model to enrich the implicit version model used by standard *VCSes*. However, they exploit that to facilitate the tailoring of their proposed *VCS* to specific application domains. That is, they extend their base version control history model to support a more fine grained versioning of specific files. For example, on top of that, they built MolhadoRef, a *VCS* that supports the capturing and versioning of the semantics of Java program entities and refactoring operations that were performed on them. Our focus, on the other hand, is not on building a new, specialized *VCS* but on enhancing a standard one with pluggable analyses that can be transparently added and removed at any time.

V. Conclusions

In this paper we proposed an architectural blueprint of a new generation *VCS* that seamlessly supports software development and software (evolution) analysis. In our vision, evolution analyses should blend into *VCSes* in a transparent and lightweight way. We are confident that this could foster a broader use of evolution analyses during real software development and not just in the confines of academic research.

Based on our blueprint, we developed a first proof of concept prototype. This prototype features a stripped down version of all the presented architectural components and of two of the plug-ins we introduced earlier in Section III: The SPARQL Endpoint and the Metrics Calculator. Building on this, we intend to proceed in developing a more sound prototype to be used in a case study. This would help us to further assess the strengths and weaknesses of our approach. We use existing *VCSes*, as our purpose is to enhance the existing ones and not to re-invent the wheel. As for the *Version Control History Model*, we already have a fairly good knowledge in modeling and describing software evolution data with ontologies [16], which we will exploit and reuse for this project. The same goes for the analyses; in the next prototype we will create plug-ins out of the many analyses our group has developed throughout the years. These analyses range from OO metrics extractors, code disharmonies calculators, fine grained source code changes distillers, etc. A partial list can be found at http://titan.ifi.uzh.ch/projects/sofas.

[2]http://code.google.com/javadevtools/codepro/doc/index.html

References

[1] M. J. Rochkind, "The source code control system," *IEEE Trans. Softw. Eng.*, vol. 1, no. 4, pp. 364–370, 1975.

[2] B. Berliner, "CVS II: Parallelizing software development," in *Proc. USENIX Winter 1990 Technical Conference*, 1990.

[3] W. F. Tichy, "RCS - A System for Version Control," *Softw. Practice and Experience*, vol. 15, no. 7, pp. 637 – 654, July 1985.

[4] A. Zeller, "The future of programming environments: Integration, synergy, and assistance," in *Future of Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 316–325.

[5] M. Dean and G. Schreiber, "OWL 2 web ontology language," W3C Recommendation, 27 October 2009, http://www.w3.org/TR/owl2-overview/.

[6] E. Prud'hommeaux and A. Seaborne, "SPARQL query language for RDF," W3C Recommendation, 15 January 2008, http://www.w3.org/TR/rdf-sparql-query/.

[7] M. Lanza, R. Marinescu, and S. Ducasse, *Object-Oriented Metrics in Practice*. Heidelberg, Germany: Springer, 2005.

[8] V. Basili, L. Briand, and W. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Trans. Softw. Eng.*, vol. 22, no. 10, pp. 751 –761, 1996.

[9] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[10] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Trans. Softw. Eng.*, 2005.

[11] H. Gall, B. Fluri, and M. Pinzger, "Change analysis with evolizer and changedistiller," *Software, IEEE*, vol. 26, no. 1, pp. 26 –33, 2009.

[12] M. Kim, D. Notkin, and D. Grossman, "Automatic inference of structural changes for matching across program versions," in *Proc. Int'l Conf. Softw. Eng.*, 2007.

[13] R. Robbes and M. Lanza, "Spyware: a change-aware development toolset," in *Int'l. Conf. Softw. Eng.*, 2008.

[14] T. Nguyen, "Object-oriented software configuration management," in *22nd International Conference on Software Maintenance (ICSM)*, 2006, pp. 351 –354.

[15] J. Tappolet, "Semantics-aware Software Project Repositories," in *Europ. Semant. Web Conf., Ph.D. Symposium*, 2008.

[16] G. Ghezzi and H. C. Gall, "Sofas: A lightweight architecture for software analysis as a service," in *Proc. Conf. Softw. Architect.*, 2011.

[17] D. Beckett and J. Broekstra, "SPARQL query results XML format," W3C Recommendation, 15 January 2008, http://www.w3.org/TR/rdf-sparql-XMLres/.

[18] M. Würsch, G. Ghezzi, G. Reif, and H. C. Gall, "Supporting Developers with Natural Language Queries," in *Int'l. Conf. Softw. Eng.*, 2010.